Lab 4: Introduction to JavaFX

This lab will give you a taste of building your own graphical applications using JavaFX. First you'll use Stages, Scenes, Panes, Rectangles, and more to create an application that visually matches our mock-up shown below. Next, you'll write EventHandler s that allows your application to respond to user input. Andy’s Graphics lectures will be a useful reference throughout this lab -- if it isn't fresh in your mind, it might help to review the lecture before you begin, or at least have it up on the screen while working.

Part 1: Building a GUI with JavaFX Panes

Goal: Create a JavaFX application that matches the above mock-up that lets users click on the colored rectangles and change their colors to a random new color. The quit button at the bottom should quit the application.

Before we begin, it’s important to note that when using large code libraries, you will definitely still run into bugs, but they might look different than in projects past. When a bug arises at runtime in a project that uses JavaFX, it will often have a long stack trace because errors from your code will be caught and then bubbled up through the JavaFX classes until it is actually thrown somewhere in the JavaFX code! Please take a moment to read the Debugging section in our JavaFX Guide for how to handle and debug this.

Note: At some point in your coding experience, it may seem like bugs are coming from the JavaFX library, but it is a well-tested library! It is way more likely that your code is subtly not performing correctly.
Getting Started

First, you’ll be building the GUI pictured above from scratch!

- Run the script cs015_install lab4 to install the stencil code for this lab.
- Open up the lab4 directory in Atom. You should see two stencil files: App.java and Constants.java. Open up App.java!

The App class will be the top-level class for your whole program. Its job is to set up the outermost graphical container (a Stage).

- In the start(...) method, set a title for your Stage passed in as a parameter to your start method using the Stage class’ setTitle(String s) method.
- Try compiling and running the program. Uh oh-- where’s the Stage? It turns out that a Stage won’t show up unless it’s told that it should be shown. Call the show() method on your Stage to make sure it shows up.
- Run the program again. You should see a small Stage pop up in the top-left corner of your screen with a grey background. You can resize the Stage by clicking and dragging its bottom-right corner.

Adding Panes

Next, we want to add some content to our Stage. We’re going to be instantiating and positioning a bunch of objects. Our Stage is just a generic container, and we don’t want it to have to worry about the details of our particular app. Instead, we’re going to create a class that will deal with all the details at a high level—let’s call it “PaneOrganizer”. Our organizer is responsible for keeping track of the Panes we will be using in our program; a Pane that contains Rectangles and a Pane that contains our label and button.

- Create a new class within the lab4 package. Name this class “PaneOrganizer”. Don’t forget that even though the name of the class is PaneOrganizer, the name of the file should be PaneOrganizer.java
- In our PaneOrganizer, we’ll be creating a couple of javafx.scene.layout.Panes and filling their contents.
- First, we want an object that’s capable of laying out JavaFX scene objects in a nice, organized way. A BorderPane will give us what we want. Create an instance variable of type BorderPane in your PaneOrganizer class and instantiate it in the PaneOrganizer constructor.
- Now write a method with the signature “public PanegetRoot()” that returns the BorderPane.
• Now we can add BorderPane to our Stage. Go back to the file App.java. In the start() method, instantiate a PaneOrganizer and add the BorderPane you’ve just created to your Scene by calling:

```java
PaneOrganizer organizer = new PaneOrganizer();
Scene scene = new Scene(organizer.getRoot());
primaryStage.setScene(scene);
```

• Note: you’ll notice that instead of directly adding the PaneOrganizer’s root Node to the Stage, we add the PaneOrganizer’s root Node to a new Scene, then add that to the Stage. You can think of a Scene as being a container for all GUI items. In CS15, you’ll only need one Scene per application.

• Note: Make sure you do all of this before the line where you show the Stage!

Are you there, Pane?

If you run the program now and expand the Stage, it looks like nothing has changed. How do we know that our BorderPane is even there? Let’s make sure everything’s working by giving our BorderPane a background color by calling the method setStyle() on it.

• Panes rely on CSS for much of their styling, so setting the background color to orange (#FFA500)* can be written one of two ways:

  ○ _pane.setStyle("-fx-background-color: orange;”);
  ○ _pane.setStyle("-fx-background-color: #FFA500;”);

• Set the background color of your BorderPane to orange in the constructor of PaneOrganizer.

• Now, when you run the program and expand the Stage by clicking and dragging, the window should be filled in orange. That means our BorderPane is displaying and everything is working properly so far.

• If you’re not seeing orange, you’ve got some debugging to do!

  ○ **TIP:** If you get an error that reads "unmappable character for encoding ASCII", try re-writing the code rather than copying it from the pdf.

*Note: CSS colors are represented by a “#” character followed by six **hexadecimal** digits or a lowercase string. For more information, see [this page](#).

Create and Size Sub-Panes
Now it’s time to create the sub-Panes we’ll need, add them to the BorderPane in the proper positions, and fill their contents according to our specification. Let’s make the top Pane, which contains the rectangles, first.

**Review: Private methods and classes**

Let’s do a quick review on the purpose of private methods and classes. In Java, there’s an important concept called “encapsulation.” The idea of encapsulation is that your code is only visible to the classes that need it. Most classes are public because we want to be able to instantiate them from everywhere, but their instance variables are private because we only want that particular class to modify them. Encapsulation helps give you more control over your code and helps you avoid a lot of particularly nasty bugs.

We’ve seen private instance variables, but we can also write private methods too! If we’re interested in factoring out some common code within a class, we can use helper methods.

```java
public class Grinch {
    // constructor code elided

    public void ruinChristmas() {
        // additional code elided
        this.stealPresents();
    }

    public void makeChildrenCry() {
        // additional code elided
        this.stealPresents();
    }

    private void stealPresents() {
        // implementation elided
    }
}
```

In the above case, we might want to factor out the “stealPresents” code that appears throughout the class. However, no one else needs to call Grinch’s stealPresents(). To prevent any bugs that might come from unintended use of stealPresents(), we make the method private. Because of this, the following code would NOT work:

```java
public class ToyStore {
    public ToyStore() {
        Grinch grinch = new Grinch();
        grinch.stealPresents(); //Will not work!
    }
}
```
When you only need to reference an object from within one particular class, it's cleaner to use *private classes*. They help encapsulate your code. They have direct access to the instance variables and methods of the class that contains them, which can come in very handy. When designing a program, you should carefully consider which classes should be public and which can be private.

Talk to your neighbor about the difference between public and private methods, and come up with one more private method that the Grinch could have in the `ruinChristmas()` method.

- Write a new private method in your `PaneOrganizer` called `createRectsPane()` that creates an instance of the `Pane` class and adds it to your `BorderPane`. This method should not return anything.

- At the beginning of this method, create a new `Pane` (call it `rectsPane`) and set its size using the `setPrefSize()` method, passing in the dimensions given in the `Constants` class for with width and height.

- Set the background color for the `rectsPane` in the same way you did for the `BorderPane`, but this time color it white (#FFFFFF).

- **Remember:** add the `Pane` you just made to your `BorderPane` using the `setTop(…)` method!

  *Note: If you are doing this lab over ssh, the initial size of the stage may be incorrect--drag the bottom-right corner to expand the window.*

- In the constructor for your `PaneOrganizer` --after instantiating your `BorderPane`-- call your `createRectsPane()` method.

- Run your program-- you should now see the top `Pane` show up!

---

**Checkpoint 1:** Compare your `PaneOrganizer` class with your neighbor!

---

**Adding Rectangles**

Now that we know that our top `Pane` has been added, let's add some `Rectangle`s to it. We want special `Rectangle`s that will support mouse clicks to change color. Let's make a new class called `ClickableRect`, which will contain a `Rectangle` and give it some special capabilities.

**Thinking about Program Design:**

Let's now consider what design options we have for the `ClickableRect`s and the `Pane`.  

---
Note: read through all options before coding any of them.

Remember: In the following designs, we refer to a Rectangle as a Node. This is valid because due to JavaFX’s inheritance structure, a Rectangle is a Node, and a Pane graphically contains Nodes. This is also laying out good practices for projects to come because we will likely want to generalize our methods to deal with composite shapes.

1. We can associate ClickableRect with the Pane by passing the Pane into the ClickableRect constructor and storing our reference to the Pane. Then, we can write methods to add or remove ClickableRect’s contained Rectangle to the associated Pane.
   // somewhere in ClickableRect.java
   
   public ClickableRect(Pane pane) {
       _pane = pane;
       // code for rectangle instantiation elided
   }
   
   public void addNode() {
       _pane.getChildren().add(_rectangle);
   }
   
   // in this lab, we will never need to remove our Nodes from the Pane, but *hint* you will have to do this in projects to come.
   public void removeNode() {
       _pane.getChildren().remove(_rectangle);
   }
   
   This may be the simplest strategy, but as we know, Java programs should be written on a need-to-know basis. Does the whole ClickableRect need to know about (or have access to) the Pane and all its public methods? The answer is no! This indicates that there may be better design choices out there.

2. We can write addNode(Pane p) and removeNode(Pane p) methods in ClickableRect. Within these methods, we can call p.getChildren().add(<our instance of Rectangle>). This design can be extended to composite shapes by using subsequent add(Node) calls or using the addAll(Node…) method.
   // somewhere in ClickableRect.java
   public void addNode(Pane p) {
       p.getChildren().add(_rectangle);
   }
This is better encapsulation than in Option 1 because it only gives access to the Pane in two of ClickableRect's methods. However, if we put Option 2 into plain English it would go something like this: "when the PaneOrganizer wants to add a ClickableRect to its Pane, it will pass its Pane to the ClickableRect and the ClickableRect will add all its Nodes to the Pane". This sounds a little convoluted and not as encapsulated (need-to-know) as possible! Let's try a third option.

3. Expose the contained Rectangle in ClickableRect to the class which contains the ClickableRect instance. We can do that by simply writing a getter, getNode(), which would return ClickableRect’s Rectangle. In PaneOrganizer (or whichever class is containing your object in future projects) you would call getNode() on the ClickableRect instance and add or remove that directly from the Pane’s children list.

```
// somewhere in ClickableRect.java
public Node getNode() {
    return _rectangle; // Rectangle is a Node (Polymorphism!)
}
```

```
// somewhere in PaneOrganizer.java we call:
-pane.getChildren().add(_clickableRect.getNode());
```

What does this design sound like in plain English? “When the PaneOrganizer wants to add a ClickableRect to its Pane, it will get the Node from ClickableRect and add it to its Pane”.

**Formal Foreshadow:** Option 3 will be our suggested design beginning next week when we learn more about the ArrayList class. ArrayLists will allow us to make a collection of all Nodes. Conveniently, the "addAll" portion of p.getChildren().addAll(...) can take in a ArrayList of Nodes as a parameter. This will be great for composite objects because we can store all the object’s Nodes in one place; at this point in the class, without ArrayLists, if we were dealing with a composite object, we would have to write individual getters for each Node--that is tedious.

It will be up to you to choose the best design for your projects; we may deduct for sub-optimal designs! For the sake of practice, we will continue on with Design Option 3 for this lab:

- Create a class called ClickableRect.
- In its constructor, initialize an instance variable of type Rectangle by instantiating a Rectangle.
- Because all ClickableRects will have the same size but different locations and
colors, we can specify the location and color as arguments to the `ClickableRect` constructor!

- Modify the `ClickableRect` constructor to take in a `double` for the x location, a `double` for the y location, and a `Color`.

  **Note:** the above mentioned `Color` is different from CSS colors.

- Then, set the size of your `Rectangle` using the width and height constants provided in the `Constants` class, as well as its location and color using the values from the `ClickableRect`'s constructor. Refer to the [Graphics II lecture](#) or [JavaFX Shapes Documentation](#) for more information about how to accomplish this.

- To align with Design Option 3, write a method to get `Rectangle` out of the `ClickableRect` class so we can add it to the scene graph. Use the signature “public `Node` getNode()” that returns the `Rectangle` you just instantiated.

- Go back to the `createRectsPane()` method.

  - Declare and instantiate 3 `ClickableRecs` called `leftRect`, `centerRect`, and `rightRect`, setting the location of each (using the values we've provided in the `Constants` class) and the color via the constructor.

- Run your program - where are your `ClickableRect` s? Much like your `RectsPane` not showing up without being added as a child of the `BorderPane` by calling `setTop()`, we need to add the `ClickableRect`s to the `Pane` as elements of the scene graph.

- To add the `Rectangles` as children of the `Pane`, we need to get the list of children belonging to the `Pane` you made earlier in `createRectsPane` and add them all to it.

  - After instantiating your Rectangles, call `rectsPane.getChildren().addAll(leftRect.getNode(), centerRect.getNode(), rightRect.getNode());`

- Try running your application again - this time, the `Rectangles` should all show up!
Creating the Bottom Pane
Let's create a `labelPane` with a label and a quit button.

- Much like you did with `createRectsPane()`, write a method called `createLabelPane()` in the `PaneOrganizer` class that creates and adds a `Pane` using the `BorderPane`'s `setBottom(...)` method.
- Call this method in the constructor of your `PaneOrganizer`.

Adding contents to the Bottom Pane

- Back in `createLabelPane()`, declare and instantiate a `Label` and a `Button`.
  - Remember that your button should say something on it! Hint: `Button` can take in a string as an argument in its constructor.
- Then, add them to the `labelPane`'s list of children.
- Run your program - the label and button should show up, but they'll be stacked on top of each other, and smashed against the left-most edge of the `Pane`!

Much like we needed the `PaneOrganizer` to contain a `BorderPane` to get special layout capabilities, we need to change the class of the `Pane` created in `createLabelPane()`.

- Take a look at the classes that subclass the `Pane` class again, looking for one that might give you vertically stacked layout capabilities
- Did you read through the documentation? Great. Now that you're more familiar with
JavaFX, you’ll know that a VBox will give you what you need.

- Change the type of your labelPane from Pane to VBox.
  
  VBox labelPane = new VBox();

- Then, to get the items in the VBox to align themselves along the center of the pane, call labelPane.setAlignment(Pos.CENTER); right after instantiating it.

- Run your program - Everything should look fine visually, but you’ll notice that clicking on the button won’t do anything! Next up - responding to user input.

Checkpoint 2: Call a TA over to check your program!

---

**Part 2: Responding to User Input**

You’ve already seen EventHandlers in lecture-- if you add an EventHandler to a component like a Button, it will “listen” for Events (like button presses). Every time it detects an event, its handle method will execute. By writing your own EventHandler that implements this method, you can tell your program how to respond when the user presses the button.

Here’s an example of a simple MouseEvent handler:

```
public class ClickHandler implements EventHandler<MouseEvent> {

    @Override
    public void handle(MouseEvent e) {
        System.out.println("Click!");
        e.consume();
    }
}
```

To add a ClickHandler to a component (let’s say we have a Pane named myPane), we could say:

```
myPane.addEventHandler(MouseEvent.MOUSE_CLICKED, new ClickHandler());
```
The first argument to the `addEventHandler` method specifies what kinds of actions it should listen for. Other examples of `MouseEvent` can be found [here](#).

In lecture, we used `setOnAction(EventHandler<ActionEvent>)`. This method is only useful for buttons. To register other `EventHandler`s, use `addEventHandler()`. For more information, go [here](#).

**Note:** We need to call `event.consume()` when dealing with `Events` in our `EventHandler`s, because without it, the event will “travel” up the scene graph. For example, if a `Pane` contains a `Button`, and they *both* have an `EventHandler` for a mouse click installed, when the click is first registered on the button, it’ll call the `Button`’s `EventHandler`’s `handle()` method, then the containing `Pane`’s `EventHandler`’s `handle()` method, and the `EventHandler` on the `Pane`’s parent, and so on. **As a rule of thumb: Always remember to `consume()` your Events!**

### Setting up an `EventHandler`

- We want to add a `ClickHandler` to our `ClickableRect` class - because it’s specific to the `ClickableRect`s, it can be a private class!
  - Start by copying our `ClickHandler` code above into your `ClickableRect.java` file, making sure it’s part of the `ClickableRect` class.
  - Change the visibility of the `ClickHandler` to `private`.
  - Once again, you’ll need to import several classes. If you try to run the program as is, the compiler won’t know which which “EventHandler” or “MouseEvent” you want! Look up the specific names for the JavaFX versions of the classes you need - if you need help, call a TA over!

- Back in the `ClickableRect` constructor, after you set the `Rectangle`’s position and color, add a new `ClickHandler` to your `Rectangle`.
- Run the app. If your `EventHandlers` are set up correctly, then every time you click on any of your `Rectangle`es you should see “Click!” printed out to the console. Once this is happening, move on to the final part of the lab.
- **Note:** make sure that your `Panels` are added to the scene graph before your `EventHandlers`.

Next, we’re going to modify the `ClickHandler` so that instead of just printing out “Click!” it randomly changes the color of the `Rectangle` you clicked on.
Before we begin, let’s take a moment to think critically about the role of the \texttt{this} keyword. The lesson we’re about to learn will be helpful when working in private inner classes like an \texttt{EventHandler}.

\texttt{this} refers to the instance of the class that we are currently in—the class whose curly braces the \texttt{this} lives in. When a line of code is evaluated in a private inner class (which though private and inner, is indeed a class), \texttt{this} will refer to the instance of the private inner class. So in terms of our code, when inside \texttt{ClickHandler}’s curly braces, \texttt{this} refers to \texttt{ClickHandler}.

So when we want to refer to the outer class, we can just tell the compiler to look at the outer class! This is much like method resolution: the Java Compiler first tries to resolve the \texttt{this} reference by first looking in the class we’re currently in, so by saying \texttt{ClickableRect.this}, the compiler actually knows to resolve \texttt{this} to be the \texttt{ClickableRect} instance.

\textbf{Note:} You may only want to use the format \texttt{<class name>\_this} when dealing with \textit{outer classes from inner classes}. Otherwise, it is unnecessary/bad style.

\begin{center}
\textbf{Customizing your \texttt{EventHandler}}
\end{center}

- Write a method in your \texttt{ClickableRect} class to generate a random color and set the color of the \texttt{Rectangle} to that color.
  - The \texttt{Color} class has a static method called \texttt{rgb()} that takes in three \texttt{ints}, one for each of the red, blue, and green values that the new color should have, and returns an instance of the \texttt{Color} class.
  - To generate a random number, you’ll want to use \texttt{Math.random()}. If you need a reminder on how to adjust this method’s output for your desired range of values, consult the \texttt{Graphics I lecture slides}!
  - Then, using the new \texttt{Color} instance you get, set the \texttt{Rectangle} to that color using its \texttt{setColor()} method.

- Now we want to edit the code in the \texttt{handle()} method of the \texttt{ClickHandler}. Instead of printing out “Click!”, call the color changing method on the \texttt{ClickableRect} it has been added to.

  \textbf{Hint:} if you’re confused on how to reference one of the \texttt{ClickableRect}’s methods from within the \texttt{ClickHandler}, reread the information above this action box.

- Run your program and start clickin’ on rectangles! If everything’s working right, each rectangle should take on a new random color whenever you click on it.
Finishing up: Adding functionality to your quit button

- You're almost finished! At this point, everything but the quit button should be functional.
- Using what you've learned so far about adding EventHandlers, go ahead and make a new handler that quits your app by calling `Platform.exit();`
  - When exiting java programs without JavaFX, you can call `System.exit(0);`
  - Remember: add an EventHandler to a button using `setOnAction`, not `addEventHandler`!
- Add an instance of your new handler to your button inside the `createLabelPane()` method
- Run your app, and test that newly functional button out!

Extra practice: Adding reset functionality

Note: this part isn't required, and you won't get extra credit for doing it, either. But it will be extra credit in future assignments! If there's time left in the lab or you're feeling ambitious on your own time, this will be a good exercise in what you've learned in this lab. It only relies on concepts you've already learned, so we won't hold your hand through this part!

- Somehow, add functionality to reset the Rectangles to their original colors. This can be via another button, clicks on some arbitrary area on your app, etc.

Javadocs for JavaFX

Last week, you were introduced to the Javadocs, which document all of the built-in Java classes. Many of the built-in classes you will work with in CS15 will be JavaFX classes, and we strongly encourage you to refer back to the documentation when working on graphical applications. Another fantastic resource we have created for you is the JavaFX Shapes Documentation, which is a simplified version of the Javadocs for JavaFX shapes. It uses less technical language than the original Javadocs, but also has slightly more limited information. You may find this documentation useful for Cartoon!
Checkpoint 3: Call over a TA once everything is working in order to get checked off!